

# OpenMPI — OpenMP like tool for easy programming in MPI

Taisuke Boku<sup>1</sup>, Mitsuhsa Sato<sup>1</sup>, Masazumi Matsubara<sup>2</sup>, Daisuke Takahashi<sup>1</sup>

<sup>1</sup> Graduate School of Systems and Information Engineering, University of Tsukuba

<sup>2</sup> Fujitsu Laboratories

{taisuke,msato,daisuke}@cs.tsukuba.ac.jp, matz@jp.fujitsu.com

## 1 Introduction

MPI has been still used for programming on wide area of applications and various sizes of problems for parallel programming based on distributed memory architecture though it requires hard labor for application programmers. Since it requires SPMD model programming, it is hard for a user to modify his original sequential program incrementally to MPI version. Of course, it also allows a user to freely describe his program in any style with communication spaghetti for very complicated interaction among parallel processes. However, a large class of scientific applications requires not so complicated communication on data and control on processes, especially for simple data parallel programming.

One of the big reasons of the success in OpenMP is that it allows incremental parallelization for the original sequential code adding appropriate directives. It helps application programmers who are not familiar to parallel processing as well as they can concentrate to the parallelization and optimization on the most time-consuming part in the target codes.

In this research, we propose a programming tool named OpenMPI for OpenMP-like incremental parallelization based on MPI scheme. A user can add directives to his original code to control parallel process invocation and communication among them automatically. The data distribution and collection among parallel processes are semi-automatically performed with invocation of parallel processes and necessary communication functions. It also supports work sharing on parallel loop like OpenMP with auxiliary functions for local index conversion and drawing out information on parallel processes. Moreover, we involve extended functions to control the parallelism dynamically as well as automatic load balancing, introducing a special user-level library named ALB (Automatic Load Balance).

Although OpenMPI limits the flexibility on MPI programming, a wide range of scientific applications may be covered and a restriction with simple notation will greatly help application programmers who want to simply paral-

lelize their target codes. We are currently designing the basic feature of OpenMPI, and will describe several design issues and preliminary specification of it.

## 2 Motivation

MPI is the most common API for parallel programming on machines with the distributed memory architecture. It supports very flexible parallelization paradigms based on message passing and there is a large set of standard functions included in version 2.0. Although a user can describe his program freely in much optimized style, most of functions are not familiar for typical scientific programmers. In NAS Parallel Benchmark 2.0, for instance, whole set of benchmarks are described only with 26 of MPI version 1.0 functions.

Domain decomposition is the most common scheme for large scale parallel scientific programming on machines with distributed memory architecture. A large set of programs can be described as very simple MPI programs which feature array data distribution, parameter broadcasting, scalar or vector reduction, etc. In such a simple program, however, the user has to describe bothering notation with long arguments to call MPI functions as well as array index conversion which often causes trivial human error.

To help poor scientific programmers who require simple and routine parallelization, we propose a new programming tool named OpenMPI based on the concept of OpenMP, that is, incremental parallelization with simple directives inserted to the original codes. Although the freedom of description is limited by such a style, we believe its feature is enough to describe a large set of programs.

For programming on distributed memory systems, various concepts and systems have been proposed and implemented. HPF[1] is an aggressive and challengeable system for automatic parallelization and data distribution for flexible programming. However, its *template*-based data distribution is too complicated for novice parallel programmers. Its automatic parallelization feature based on owner computation rule is also difficult to exploit the potential parallelism

and performance of the system.

UPC[2] is another challenge for efficient parallel processing on distributed shared memory architecture to provide straight-forward programming for traditional C language. It supports explicit classification of memory attribute of *private* and *shared* to exploit high performance based on process-data annotation and access locality.

There are a number of researches on generic software DSM (distributed shared memory) systems such as TreadMarks[3] or SCASH/Score[4]. Most of them are based on page-base state management and invalidation-based protocol. Since they require the help of operating system, its performance strongly depends on the application and the system overhead is not negligible on some class of applications.

Compared with these systems, we simply aim to provide simple programming tool for MPI, that is, the user does not expect fully automatic parallelization or the correctness of execution guaranteed by the system. It relieves the painful and bothering programming on MPI as like as OpenMP relieves the programmers from bothering and easy programming error on thread programming.

### 3 Basic requirements for parallel programming based on domain decomposition

We consider the domain decomposition as the basic parallelization concept in our system. It means that the target problem space is represented as multiple dimensions of data arrays and they are decomposed into multiple chunks to be mapped on to parallel processes. The most time consuming part of the program is the computation on these distributed array data which can be completely parallelized among processes. At certain timing, local or global (collective) communication is required to maintain the consistency and system-wide information sharing.

The basic features required for this purpose are summarized as follows:

**array distribution:** The arrays to represent the problem space have to be automatically divided into chunks according to the number of parallel processes. To simplify the workload distribution, the target array have to be sliced orthogonally to the selected dimension(s). A decomposed array chunk is allocated to a process and it will not be changed.

**index conversion:** The global array indexing to the original (sequential) code has to be converted into local array indexing according to the array distribution. By simplified array slicing described above, it can be performed automatically for linear indexing.

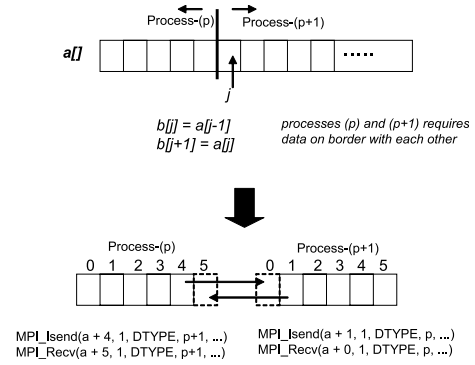


Figure 1. Cross data reference on border elements

**data consistency:** To support cross-indexing over neighboring elements in an array at the border of distributed array (see Figure 1), automatic data exchanging to maintain data consistency among processes is necessary. It appears as “nearest neighboring communication” in typical MPI programs. To avoid the overhead for dynamic check on data access, the user has to be responsible to the region and timing to keep data consistency.

**data reduction:** To share global information among processes, collective communications for data reduction are required at certain timing. The target variable and the timing must be explicitly specified by the user.

**data synchronization:** Since MPI program is described in SPMD manner, even a scalar variable has to be duplicated among parallel processes. A “sequential” computation on such a variable is performed simultaneously in all processes. If the computation depends on some local information kept in a certain process, the computation result must be shared by all processes after the computation. This feature is named “synchronization”, and it must be explicitly specified by the user.

**auxiliary functions:** In OpenMPI, we allow the user to describe explicit MPI function calls as well as the automatically generated MPI code by OpenMPI compiler. However, the user has to understand the data distribution, the size of local array and index range to be assigned to each process. To assist such an explicit data access, miscellaneous functions to provide auxiliary information are required. For instance, index conversion functions to convert global and local array indices help to access an element of local array from a global index written in the original sequential code.

## 4 OpenMPI directives

The programming with OpenMPI is in similar manner with OpenMP programming. The user inserts appropriate directives to create parallelized version of his program. Here, he must understand that he is describing MPI program with very simple notation, that is, the compiler and run-time system do not provide fully automatic parallelization nor complete shared address space among parallel processes. The user must understand the effect of these directives and be responsible to the result of parallel execution under the rule of OpenMPI directives.

The followings are the typical directives defined in current designing. All directives follow to the notation “**#pragma ompi**”. Terms in bold font are reserved words in OpenMPI and terms in italic font are user defined parameters.

- **distvar** (*dim=dimension, sleeve=sleeve\_size*)  
The array declared in the next line is distributed on parallel processes. The option **dim** represents the sliced dimension when the array has more than one dimension<sup>1</sup>. The option **sleeve** represents the sleeve size of border elements which may be exchanged automatically (see Figure 1). If the **sleeve** option is omitted, there is no automatic element exchanging on border for that array.
- **global**  
The variable declared in the next line is assumed as global one, that is, the content of the variable will be maintained to the same value among all processes. Since the target architecture is the distributed memory system, it is impossible to make the variable physically coherent. Another directive **sync\_var** represents the point to copy the data.
- **for** (**reduction**(*operator : variable*))  
The for loop at the next line is parallelized and all distributed array data indexed by the control function of **for** statement and its linear expression are referred only within the assigned region to the process according to slicing feature of **distvar** directive. The option **reduction** causes the collective communication on *variable* with *operator* after the end of the loop.
- **sync\_sleeve** (*var=variable list*)  
At this point, the sleeve data of specified distributed array are exchanged to maintain the correctness of neighboring data reference. If **var** option to specify the name of array is omitted, all distributed array data are processed.

<sup>1</sup>Currently, only one dimension can be sliced, and there is no feature of multiple-dimension slicing.

- **sync\_var** (*var=variable list, master=node\_id*)  
The contents of the specified global variable (scalar or array) among all processes are synchronized by copying a master data to others. Process-0 is assumed as the default master if no **master** option is specified. Otherwise, the specified node is the master.
- **single** (*master=node\_id*)  
The next block is executed by only one process as the representative of all processes. The process to execute the following block is specified by **master** option. Otherwise, process-0 is assumed as the master.

## 5 Sample code

Figures 2 and 3 show a sample code of OpenMPI program in C. The target problem is the explicit solution of 2-dimensional Laplace equation with convergence check. 2-D arrays *u* and *nu* are sliced at the second dimension (*X*-dimension, *dim=1*) as shown in Figure 4.

```
#include <openmpi.h>
#include <stdio.h>

#define XSIZE 128
#define YSIZE 128
#define EPS 1.0e-2

#pragma ompi distvar (dim=1,sleeve=1)
double u[YSIZE + 2][XSIZE + 2];
#pragma ompi distvar (dim=1)
double nu[YSIZE + 2][XSIZE + 2];

main()
{
    int i, j, k;
    double res1, res2, tmp;
    int itr;

    /* initialization */
    for(i = 1; i <= YSIZE; i++)
#pragma ompi for
        for(j = 1; j <= XSIZE; j++)
            u[i][j] = 1.0;
#pragma ompi for
    for(j = 1; j <= XSIZE; j++){
        u[0][j] = 10.0;
        u[YSIZE+1][j] = 10.0;
    }
    for(i = 1; i <= YSIZE; i++){
        u[i][0] = 10.0;
        u[i][XSIZE+1] = 10.0;
    }
    /* iteration */
    itr = 1;
    while(1){
#pragma ompi sync_sleeve
        for(i = 1; i <= YSIZE; i++)
#pragma ompi for
            for(j = 1; j <= XSIZE; j++)
                nu[i][j] = (u[i-1][j] + u[i+1][j]
                    + u[i][j-1] + u[i][j+1]) / 4.0;
        /* convergence check */
        res1 = 0.0;
```

Figure 2. Example code (Laplace Eq.) (1/2)

```

for(i = 1; i <= YSIZE; i++){
    res2 = 0.0;
#pragma ompi for reduction(+:res2)
    for(j = 1; j <= XSIZE; j++){
        tmp = (nu[i][j] - u[i][j]) / u[i][j];
        res2 += tmp * tmp;
    }
    res1 += res2;
}
#pragma ompi single
fprintf(stderr, "itr=%d res=%g\n", itr, res1);
/* update */
for(i = 1; i <= YSIZE; i++)
#pragma ompi for
    for(j = 1; j <= XSIZE; j++)
        u[i][j] = nu[i][j];
if(res1 < EPS)
    break;
itr++;
}

/* output */
for(i = 1; i <= YSIZE; i++)
#pragma ompi for
    for(j = 1; j <= XSIZE; j++)
        printf("%d %d %f\n", i, j, u[i][j]);
}

```

Figure 3. Example code (Laplace Eq.) (2/2)

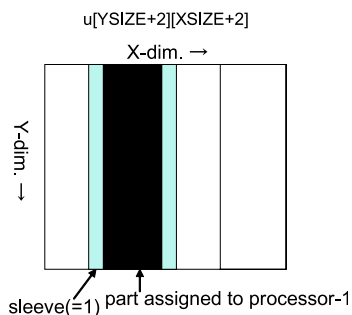


Figure 4. 2-D array distribution of  $u$

A subpart of array  $u$  assigned to each process has two sleeves on left and right edges with size=1. It means that these edges at the border of neighbors are overlapped with neighbors' data on corresponding edges with the width of 1. These overlapped data are automatically exchanged at the point of “**#pragma ompi sync\_sleeve**” in every iteration. Local array of  $u$  is declared including these two additional elements to accept data sent from left and right neighbors, then they can be referred in the following second loop to calculate new value of  $u$  saved into  $nu$ . On the other hand,  $nu$  has no sleeve because subparts of  $nu$  are completely treated as local variable without any interaction with neighbors.

In the latter half of the big loop (while statement), a scalar variable  $res2$  is treated as reduction variable on for loop to check the convergence of the result.

## 6 Compiler

The compiler of the prototype implementation of OpenMPI is now under construction. It is implemented as a source-to-source converter to generate a complete MPI code from the target code. Figures 5 and 6 show the image of converted code of the program shown in Figures 2 and 3.

```

#include "openmpi.h"
#include <mpi.h>
#include <stdio.h>

#define XSIZE 128
#define YSIZE 128
#define EPS 1.0e-3

double u[YSIZE + 2][XSIZE + 2];
double nu[YSIZE + 2][XSIZE + 2];

/* variables inserted by compiler */
int _nproc, _myid;
int _u_idx_lower, _u_idx_upper, _u_idx;
int _nu_idx_lower, _nu_idx_upper, _nu_idx;
int _u_sleeve = 1;
int _u_dim = 1;
int _nu_sleeve = 0;
int _nu_dim = 1;

main(int argc, char **argv)
{
    int i, j, k;
    double res1, res2, tmp;
    int itr;
    int _l_j, _l_j_llimit, _l_j_ulimit;
    double _tmp_res2; /* local index var. */
    /* for reduction */

    mpi_init(&argc, &argv);
    mpi_record_var(u, 2, 1, 1, sizeof(double),
        &_u_idx_lower, &_u_idx_upper, YSIZE+2, XSIZE+2);
    mpi_record_var(nu, 2, 1, 0, sizeof(double),
        &_nu_idx_lower, &_nu_idx_upper, YSIZE+2, XSIZE+2);
    /* initialization */
    for(i = 1; i <= YSIZE; i++){

```

Figure 5. Converted code of Laplace Eq. (1/2)

```

    for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit; _l_j++)
        u[i][_l_j] = 10.0;
}
_l_j_llimit = omp_get_llimit(1, _u_idx_lower);
_l_j_ulimit = omp_get_ulimit(XSIZE, _u_idx_upper);
for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit; _l_j++) {
    u[0][_l_j] = 10.0;
    u[YSIZE+1][_l_j] = 10.0;
}
for(i = 1; i <= YSIZE; i++) {
    _u_idx = omp_convert_local(0, _u_idx_lower,
                               _u_idx_upper);

    if(_u_idx >= 0) {
        u[i][_u_idx] = 10.0;
    }
    _u_idx = omp_convert_local(XSIZE+1, _u_idx_lower,
                               _u_idx_upper);

    if(_u_idx >= 0) {
        u[i][_u_idx] = 10.0;
    }
}
/* iteration */
while(1) {
    omp_sync_sleeve();
    for(i = 1; i <= YSIZE; i++) {
        _l_j_llimit = omp_get_llimit(1, _u_idx_lower);
        _l_j_ulimit = omp_get_ulimit(XSIZE,
                                     _u_idx_upper);

        for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit;
            _l_j++)
            nu[i][_l_j] = (u[i-1][_l_j] + u[i+1][_l_j]
                          + u[i][_l_j-1] + u[i][_l_j+1]) / 4.0;
    }
    /* convergence check */
    res1 = 0.0;
    for(i = 1; i <= YSIZE; i++) {
        res2 = 0.0;
        _tmp_res2 = 0.0;
        _l_j_llimit = omp_get_llimit(1, _u_idx_lower);
        _l_j_ulimit = omp_get_ulimit(XSIZE, _u_idx_upper);
        for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit;
            _l_j++) {
            tmp = (nu[i][_l_j] - u[i][_l_j]) / u[i][_l_j];
            _tmp_res2 += tmp * tmp;
        }
        omp_reduce(OMPI_REDUCE_OP_SUM, OMPI_DATA_DOUBLE,
                  1, &res2, &_tmp_res2);
        res1 += res2;
    }

    if(omp_master()) {
        fprintf(stderr, "itr=%d res=%g\n", itr, res1);
    }
    /* update */
    for(i = 1; i <= YSIZE; i++) {
        _l_j_llimit = omp_get_llimit(1, _u_idx_lower);
        _l_j_ulimit = omp_get_ulimit(XSIZE, _u_idx_upper);
        for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit; _l_j++)
            u[i][_l_j] = nu[i][_l_j];
        /* possible, not aligned ... */
    }
    if(res1 < EPS)
        break;
    itr++;
}

/* output */
for(i = 1; i <= YSIZE; i++) {
    _l_j_llimit = omp_get_llimit(1, _u_idx_lower);
    _l_j_ulimit = omp_get_ulimit(XSIZE, _u_idx_upper);
    for(_l_j = _l_j_llimit; _l_j <= _l_j_ulimit; _l_j++)
        printf("%d %d %f\n", i,
              omp_convert_global(_l_j, _myid), u[i][_l_j]);
}
}

```

Figure 6. Converted code of Laplace Eq. (2/2)

Array variables to be distributed to all processes are declared in the same manner as the original code. Currently, only global and static variables are allowed as **distvar** objects, and the array size of converted version is as same as the original. It is because the optimal size of local array, which could be smaller than the original, depends on the number of processes to share the array, and we consider that the number of processors to be used is fixed at run time. One of the solutions is to assume the minimum number of processes at compile time and reduce the size of array.

According to the index range assigned to the process, the reference to arrays is limited between the lower and upper limits precalculated by internal functions `omp_get_llimit()` and `omp_get_ulimit()`. As shown in this code, only a linear expression of control variable ( $j$ , here) is allowed for OpenMPI *for* statement.

At the corresponding point of **sync\_sleeve** directive, an internal function `omp_sync_sleeve()` is called to exchange sleeve data between neighboring processes. Within the function, the array information (starting address, dimension, size of each dimension and sleeve size) is referred to determine the data area to be sent and received. For this purpose, all distributed array data are registered by an internal function `omp_record_var()` at the beginning of main program. The contents of these sleeves are packed into the send buffers, then they are transferred asynchronously. After receiving, all received data are unpacked into the corresponding sleeve area in the target arrays. Since we allow any dimension of the distributed array to be sliced, these exchanged area generally forms a block-stride pattern. Therefore, we need a constant size of pack/unpack buffer for each sleeve area.

## 7 Automatic Load Balancing

Since OpenMPI introduces the abstraction of parallelization instead of explicit MPI parallel programming, there is a possibility to control the parallelism not only at compile time but also at run time. Ordinary MPI code assumes that the number of parallel processes is fixed at the beginning of run time, but MPI 2.0 supports the dynamic increase/decrease of processes while executing the code. Applying this feature to our system, we can dynamically control the parallelism while iterating the loop. For instance, if we operate a big PC cluster for multiple jobs, this dynamic feature greatly enhances the total system efficiency according to the size and number of jobs to be executed. With explicit MPI programming, it is very hard to describe such a dynamic code even with MPI 2.0 feature.

OpenMPI compiler generates a pure MPI code which can be executed on generally used MPI libraries such as LAM[5] or MPICH[6]. We are also designing an optional feature to support dynamic parallelism based on LAM MPI

7.0 or higher. On LAM MPI, there is a set of low level commands to control the number of running parallel processes on user level. For easier implementation and flexibility on user interface, we apply ALB (Automatic Load Balance) tool, which is developed by Fujitsu Laboratories, to OpenMPI compiler.

ALB is developed based on dynamic parallelization control feature on LAM MPI. The tool consists of two parts: API library for extended MPI programming and user level commands to control the number of processors involved in parallel execution. Current design of OpenMPI compiler is strongly considered about the suitability of the converted code with ALB library. For example, array distribution with one dimensional slicing with sleeve data and the synchronization of global variable are directly supported by ALB. Moreover, ALB checks the increase/decrease of processes at the timing of synchronization among variables, and automatically redistributes registered arrays according to the information registered at the beginning of computation.

In this extended abstract, we have no space to describe the detail of ALB, however the combination of ALB-based OpenMPI and run-time monitoring system to check the load of the cluster provides a flexible and efficient system management method for large scale cluster.

## 8 Conclusions

OpenMPI is a new scheme to provide very easy programming method of MPI code for scientific application users. Although it limits the flexibility and freedom of programming, the concept of incremental parallelization of OpenMP is partially applied to help programmers. ALB also supports the dynamic parallelism control on OpenMPI code for efficient usage of CPU resources in a cluster system shared by multiple users and tasks. The easiness of ALB usage is greatly enhanced by introducing OpenMPI.

The system is still under construction. After the completion, we will describe various sample codes including NAS PB to demonstrate the capability of code description and parallelization of OpenMPI.

## References

- [1] <http://dacnet.rice.edu/Depts/CRPC/HPFF/>
- [2] <http://upc.gwu.edu/>
- [3] C. Amza, et. al., "TreadMarks: Shared Memory Computing on Networks of Workstations", IEEE Computer, Vol. 29, No. 2, pp. 18-28, February 1996.
- [4] <http://www.pccluster.org/score/dist/score/html/en/reference/scash/index.html>
- [5] <http://www.lam-mpi.org/>
- [6] <http://www-unix.mcs.anl.gov/mpi/mpich/>